1 Introduzione

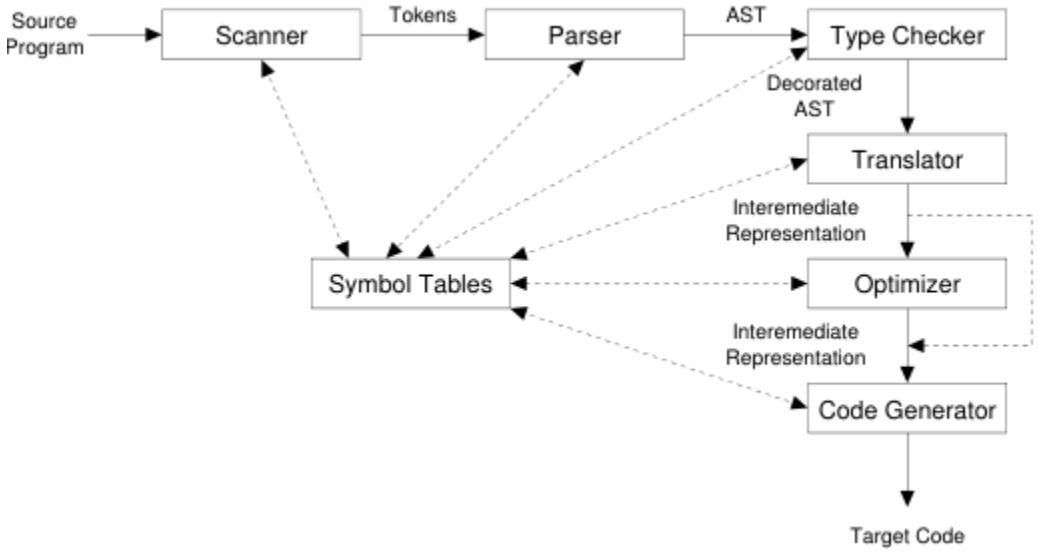
1.1 Sintassi e semantica

La definizione di un linguaggio di programmazione deve includere la specificazione della sua *sintassi* (struttura) e *semantiche* (significato). La sintassi definisce quali sequenze di caretteri sono ammesse.

1.2 Organizzazione di un compilatore

I compilatori generalmente svolgono i seguenti compiti:

- *analisi* del programma sorgente da compilare
- *sintesi* del programma destinazione



Quasi tutti i compilatori moderni sono *orientati alla sintassi*. I compilatori sintetizzano la struttura di un programma in un *abstract syntax tree* (AST), che omette dettagli superflui. Il parser costruisc l'AST usando *tokens*, i simboli elementari usati per definire la sintassi del linguaggio di programmazione. Il riconoscimento della struttura sintattica è una parte importante della *analisi sintattica*.

L'*analisi semantica* esamina il significato (semantica) del programma, basandosi sulla sua struttura sintattica.

Nella *fase di sintesi*, i costrutti del linguaggio sorgente sono tradotti in una *rappresentazione intermedia* (IR) del programma, anche se alcuni compilatori generano direttamente il codice di destinazione.

1.2.1 Scanner

Lo *scanner* comincia l'analisi del programma sorgente leggendo carattere per carattere il testo in input, raggruppando i singoli caratteri in token (identificatori, interi, parole riservate, delimitatori). I token vengono solitamente codificati e passati al parser per l'analisi sintattica. Lo scanner svolge i seguenti compiti:

- trasforma il programma in un flusso di token
- elimina informazioni superflue (come i commenti)
- processa le direttive del compilatore

Le *espressioni regolari* sono un metodo efficiente per descrivere i tokens.

1.2.2 Parser

Il parser è basato su specificazioni sintattiche formali come le grammatiche context-free. Legge i tokens e li raggruppa in frasi a seconda della specificazione della sintassi.

Il parser verifica che la sintassi sia corretta. Se incontra un errore di sintassi, riporta un messaggio di errore appropriato. Inoltre, può tentare di rimediare all'errore.

1.2.3 Type Checker

Il *type checker* controlla la *semantica statica* di ogni nodo dell'AST: verifica che il costrutto che ogni nodo rappresenta sia legale e significativo. Se il costrutto è semanticmante corretto, il type checker decora il nodo dell'AST aggiungendo informazioni di tipo.

Il type checking dipende solamente dalle regole semantiche del linguaggio sorgente.

1.2.4 Symbol tables

Una *symbol table* è un meccanismo che permette l'associazione di informazioni agli identificatori e la loro condivisione nelle fasi di compilazione.

1.2.5 Generatore di codice

Il codice intermedio prodotto da un traduttore è mappato sulla macchina di destinazione da un *generatore di codice*.

2 Un semplice compilatore

2.1 Definizione informale del linguaggio *ac*

Definiamo *ac* informalmente:

Tipi In *ac* esistono solo due tipi di dato: intero e float.

Keywords In *ac* esistono tre tipi di parole riservate: float, int e print.

Variabili In *ac* il nome della variabile è dichiarato dopo averne specificato il tipo. La maggior parte dei linguaggi di programmazine possiede regole che dettano la conversione di tipo: in *ac*, la conversione da intero a float avviene automaticamente.

Il linguaggio di destinazione è *dc*, una calcolatrice a stack che utilizza la notazione polacca inversa.

2.2 Definizione formale di *ac*

Useremo una grammatica context-free per specificare la sintassi del linguaggio e espressioni regolari per specificare i simboli del linguaggio.

2.2.1 Specifica della sintassi

Il linguaggio *ac* è specificato da una grammatica context-free (CFG), un insieme di *produzioni*.

La produzioni fanno riferimento a due tipi di simboli: i *terminali* e *non terminali*. Un terminale è un simbolo della grammatica che non può essere riscritto.

Una CFG descrive in maniera compatta e formale tutti i programmi che possono essere generati da un dato linguaggio di programmazione. Per generare un programma, si inizia da uno speciale simbolo

non terminale, detto *assioma*, che solitamente è il simbolo nella parte sinistra (LHS) della prima regola di produzione della grammatica.

La generazione procede rimpiazzando non terminali con altri non terminali o terminali, secondo le regole della grammatica. Il simbolo ε rappresenta la *stringa vuota*, che indica l'assenza di simboli nella parte sinistra di una produzione. Il simbolo $ rappresenta la fine dell'input.

2.2.2 Specifica dei tokens

La specificazione formale dei token di un linguaggio viene tipicamente compiuta con associando *espressioni regolari* ad ogni token.

2.3 Fasi del compilatore

1. Lo scanner legge il programma sorgente da un file di testo e produce un flusso di tokens. Le parole riservate vengono distinte da nomi di variabili.
2. Il parser processa i tokens, ne determina la validità sintattica e crea un AST.
3. L'AST così creata viene attraversata per creare una symbol table. Questa tabella associa tipo e altre informazioni alle variabili usate nel programma sorgente.
4. L'AST viene attraversato per eseguire l'analisi semantica. L'analisi semantica decora o trasforma parti dell'AST man mano che il significato di tali parti diventa chiaro.
5. Infine, l'AST viene attraversato per generare la traduzione del programma sorgente.

2.4 Scanning

Il compito dello scanner è quello di trasformare un flusso di caratteri in un flusso di token. dove ogni token rappresenta un'istanza di un simbolo terminale. Ogni token scandito dallo scanner possiede un *tipo* e *valore semantico*.

2.5 Parsing

Il parser controlla la validità del flusso di tokens. La *discesa ricorsiva* è una delle più semplici tecniche di parsing: ogni non terminale è associato ad una procedura di parsing che determina se il token nel flusso contiene una sequenza di token derivabile da quel non terminale.

2.6 Abstract Syntax Trees

Lo scanner e il parser svolgono la fase di *analisi sintattica* di un compilatore. L'AST contiene informazioni essenziali derivate dall'albero di parsing.

2.7 Analisi semantica

Durante l'analisi semantica vengono svolte le seguenti operazioni:

- le dichiarazioni e gli scope sono processati per costruire una *symbol table*
- i tipi definiti dal linguaggio e dall'utente sono esaminati per assicurarne la consistenza

2.7.1 Symbol tables

La costruzione della symbol table è un'attività di processamento semantico durante la quale l'AST viene attraversato per registrare tutti gli identificatori ed i loro tipi in una *symbol table*.

2.7.2 Type checking

Il linguaggio *ac* offre solo due tipi, intero e float, e tutti gli identificatori devono avere il loro tipo dichiarato. Una volta che la symbol table è stata costruita, il tipo di ogni identificatore è conosciuto, e si può verificare la consistenza dei tipi.

Durante il type checking, l'AST viene attraversato dal basso verso l'alto, ed ad ogni nodo viene applicato il metodo visitor appropriato:

- Per le costanti e i simboli, il visitor imposta il tipo del modo basandosi su contenuti dello stesso.
- Per nodi che computano valori, viene eventualmente effettuata conversione di tipo.
- Per l'operazione di assegnamento, il visitor si assicura che il valore calcolato dal secondo nodo sia dello stesso tipo del primo nodo.

2.8 Generazione del codice

L'ultimo compito del compilatore è la formulazione di codice che rappresenti le semantiche del programma sorgente. Per *ac*, il linguaggio di destinazione è *dc*, una calcolatrice basata su stack.

La generazione del codice avviene attraversando l'AST, dalla radice alle foglie (top-down).

3 Scanning

3.1 Espressioni regolari