1 Introduzione

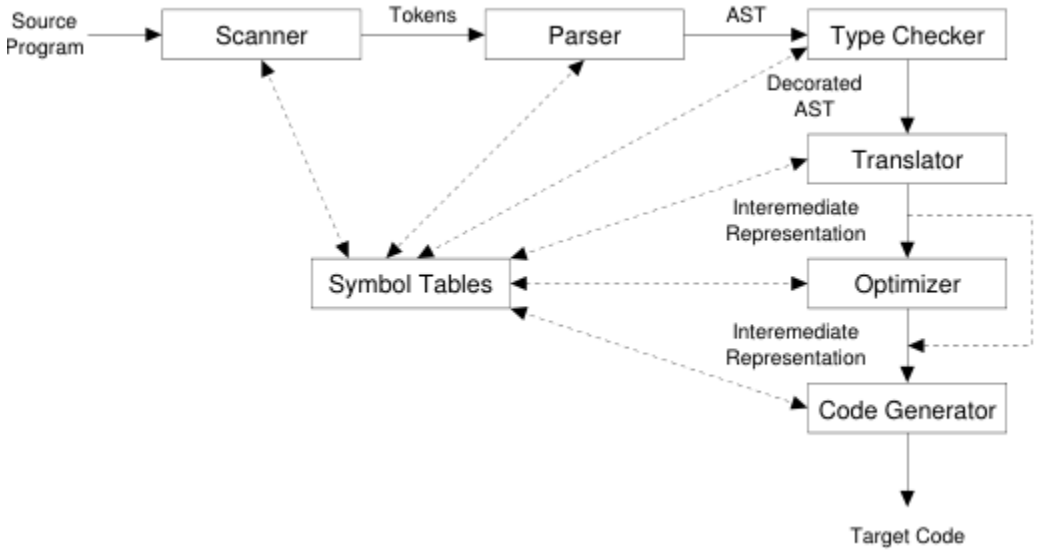
1.1 Sintassi e semantica

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